Paper Number 128

Clindex: Clustering for Similarity Queries in High-Dimensional Spaces

Chen Li, Edward Chang, Hector Garcia-Molina
James Ze Wang and Gio Wiederhold
Department of Computer Science, Stanford University

Abstract

In this paper we present a clustering and indexing paradigm (called Clindex) for high-dimensional search spaces. The scheme is designed for approximate searches, where one wishes to find many of the data points near a target point, but where one can tolerate missing a few near points. For such searches, our scheme can find near points with high recall in very few IOs and performs significantly better than other approaches. Our scheme is based on finding clusters, and then building a simple but efficient index for them. We analyze the tradeoffs involved in clustering and building such an index structure, and present experimental results based on a 30,000 image database.
Keywords: similarity search, multidimensional indexes.

1 Introduction

Similarity search has generated a great deal of interest lately because of applications such as similar text/image search and document/image copy detection. These applications characterize objects (e.g., images and text documents) as *feature vectors* in very high-dimensional spaces [6, 15]. A user submits a query object to a search engine, and the search engine returns objects that are similar to the query object. The degree of similarity between two objects is measured by the Euclidean distance between their feature vectors. The search is performed by returning the objects that are *nearest* to the query object in high-dimensional spaces.

Nearest neighbor search is inherently expensive, especially when there are a large number of dimensions. There is simply no way to build an index on disk such that *all* nearest neighbors to *any* query point are physically near on disk. (We discuss this "curse of dimensionality" in more detail in Section 2.) Fortunately, in many cases it is sufficient to perform an *approximate search* that returns many but not all nearest neighbors [2, 8, 18, 23, 24]. (A feature vector is often an approximate characterization of an object, so we are already dealing with approximations anyway.) For instance, in content-based image retrieval [6, 10] and document copy detection [5, 11] it is usually acceptable to miss a small fraction of the target objects, so it is not necessary to pay the high price of exact search.

We present in this paper a new similarity search paradigm: a clustering/indexing combined scheme that achieves approximate similarity search with high efficiency. We call this scheme *Clindex* (CLustering for INDEXing). Under Clindex, the dataset is first partitioned into "similar" clusters. Clindex can be used with a variety of clustering algorithms, but in this paper

we explore a very natural algorithm that can be considered a simplified version of the CLIQUE Algorithm [1]. To improve IO efficiency, each cluster is then stored in a sequential file, and a mapping table is built for indexing the clusters. To answer a query, clusters that are near the query point are retrieved.

Both clustering and indexing have been intensively researched, but these two subjects have been studied separately for different optimization objectives: clustering optimizes classification accuracy [1, 9, 27, 36] while indexing maximizes IO efficiency for information retrieval. Because of these different goals, the indexing schemes often do not preserve the clusters of the dataset and randomly project objects that are close (hence similar) in high-dimensional spaces onto a 2D plane (the disk geometry). This is analogous to breaking a vase (cluster) apart to fit it into the minimum number of small packing boxes (disk blocks). Although the space required to store the vase may be reduced, finding the boxes in a high-dimensional warehouse to restore the vase requires a great deal of effort.

The contributions made by this paper are as follows:

- We study how clustering and indexing can be effectively combined. Note that we are *not* proposing a new clustering nor a new indexing scheme. Instead, we are showing how a rather natural clustering scheme (using grids as in [1]) can lead to an extremely simple index that performs very well for approximate similarity searches. We believe that *simplicity* is one of the major strengths of our approach.
- We experimentally evaluate the Clindex approach, using a 30,000 image database. Our results show that Clindex achieves very high *recall*. That is, it can typically return more than 90% of what we call the "golden" results (i.e., the best results produced by a linear scan over the entire dataset) with a few IOs.
- We also compare Clindex with other traditional approaches, to evaluate how effective Clindex's clustering is, and to understand the gains achievable by pre-processing data to find clusters.
- If data does not have natural clusters, then Clindex will not improve searches. Fortunately, real datasets rarely occupy a large-dimensional space uniformly [30, 36]. Our experimental results confirm that, in the specific case of image data, significant clusters are formed, and that they can be exploited to noticeably improve search times.
- We discuss how Clindex parameters can be tuned to improve performance. We also show experimental results that provide insights into the tuning process.

The rest of the paper is organized as follows. Section 2 discusses the shortcomings of some traditional approaches employed to conduct similarity queries in high-dimensional spaces. Section 3 presents Clindex and shows how a similarity query is conducted using our scheme. Section 4 presents the results of our experiments and compares the efficiency and accuracy of our approach with that of some traditional index structures. Finally, we offer our conclusions in Section 5.

2 Background

Many tree structures have been proposed to index high-dimensional data (e.g., R*-tree [3, 16], SS-tree [35], SR-tree [22], TV-tree [25], X-tree [4], M-tree [7], and K-D-B-tree [28]). A tree structure divides the high-dimensional space into a number of subregions, each containing a subset of objects that can be stored in a small number of disk blocks. Given a vector that represents an object, a similarity query takes the following three steps in most systems [12]:

1. It performs a *where-am-I* search to find out in which subregion the given vector resides.

2. It then performs a *nearest-neighbor* search to locate the neighboring regions where similar vectors may reside. This search is often implemented using a *range* search, which locates all the regions that overlap with the search sphere, i.e., the sphere centered at the given vector with a diameter d.

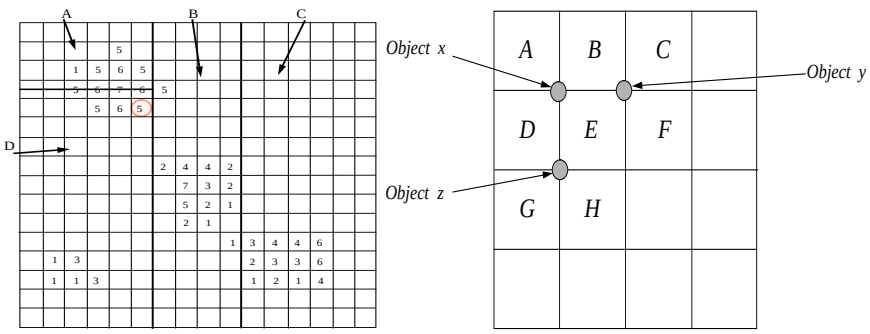
3. Finally, it computes the Euclidean distances between the vectors in the nearby regions (obtained from the previous step) and the given vector. The search result includes all the vectors that are within distance d from the given vector.

The performance bottleneck of similarity queries lies in the first two steps. In the first step, if the index structure does not fit in the main memory and the search algorithm is inefficient, a large portion of the index structure must be fetched from the disk. In the second step, the number of neighboring subregions can grow exponentially with respect to the dimension of the feature vectors. If D is the number of dimensions, the number of neighboring subregions can be on the order of $O(3^D)$ [12]. Reading in the data from all neighboring regions can thus take an exponential number of IOs. (Even if one can use intelligent schemes to cut down the search space, e.g., [29], the reduced search space is still on the order of $O(3^D)$.)

In addition to being copious these IOs can be random and hence exceedingly expensive.[1] An example can illustrate what we call the *random-placement* syndrome faced by the traditional index structures. Figure 1(a) shows a 2-dimensional Cartesian space divided into 16 equal stripes in both the vertical and the horizontal dimensions, forming a 16×16 grid structure. The integer in a cell indicates how many points (objects) are in the cell. Most index structures divide the space into subregions of equal points in a top-down manner. Suppose each disk block holds 40 objects. One way to divide the space is to first divide it into three vertical compartments, left, middle, and right, and then to divide the left compartment horizontally. We are left with four subregions A, B, C, and D containing about the same number of points. Given a query object residing near the border of A, B and D, the similarity query has to retrieve blocks A, B and D. The number of subregions to check for the neighboring points grows exponentially with respect to the data dimension.

Furthermore, since in high-dimensional spaces the neighboring subregions cannot be arranged in a manner sequential to all possible query objects, the IOs must be random. Figure 1(b) shows a 2-dimensional example of this random phenomenon. Each grid in the figure, such as A and B, represents a subregion that corresponds to a disk block. The figure shows

[1] To transfer 100 KBytes of data on a modern disk with eight KBytes block size, doing a sequential IO is more than ten times faster than doing random IOs. The performance gap widens as the size of the data transfer increases.



(a) Clustering by Indexing (b) Random Placement

Figure 1: The Shortcomings of Tree Structures.

three possible query objects x, y and z. Suppose that the neighboring blocks of each query object are its four surrounding blocks. For instance, blocks A, B, D and E are the four neighboring blocks of object x. If the neighboring blocks of objects x and y are contiguous on disk, then the order must be C, F, E, B, A, D, or the reverse order. Then it is impossible to store the neighboring blocks of query object z contiguously on disk, and this query must suffer from random IOs. This example suggests that in high-dimensional spaces, the neighboring blocks of a given object must be dispersed randomly on disk by tree structures.

Theoretical studies [2, 18, 23] have confirmed the cost of exact search, independent of the data structure used. In particular, it has been shown that if N is the size of a dataset, D is the dimension, and $D >> log\ N$, then no nearest neighbor algorithm can be significantly faster than a linear search.

2.1 Approximate Similarity Search

Many studies propose conducting approximate similarity search for applications where trading a small percentage of recall for faster search speed is acceptable. For example, instead of searching in all the neighboring blocks of the query object, the study of [34] proposes performing only the where-am-I step of a similarity query and returning only the objects in the disk block where the query object resides. Take Figure 1(a) as an example. Suppose the query object is in the circled cell in the figure, which is near the border of regions A, B and D. If we return only the objects in region D where the query object resides, we miss many nearest neighbors in A.

Many nearest-neighbor search techniques have been proposed to do similarity search in high-dimensional spaces. However, due to the curse of dimensionality, many recent studies suggest doing only ε-approximate nearest neighbor searches, for $\varepsilon > 0$ [2, 8, 18, 23]. Let d denote the function computing the distance between two points. We say that $p \in P$ is an ε-approximate nearest neighbor of q if for all $p' \in P$ we have $d(p,q) \leq (1+\varepsilon)d(p',q)$. Indyk and Motwani [18, 19] propose ε-approximate algorithms that enjoy improved performance bounds over Kleinberg's algorithms [23]. But the study of Kushilevitz et al. [24] points out that the recall of the algorithms proposed in [19] can be low since the search distance can be increased from a very

small δ to something that is much larger than $(1+\varepsilon)\delta$. To improve recall, a follow-up study of [19] builds multiple *locality-preserving* indexes on the same dataset [17]. This is analogous to building n tree indexes on the same dataset, and each index distributes the data into data blocks differently. To answer a query, one retrieves one block following each of the indexes and combines the results. Obviously, this approach achieves better recall than is achieved by having only one index. But in addition to the n times pre-processing overhead, it has to replicate the data $n-1$ times to ensure that sequential IOs are possible via every index.

Two other approaches have also been proposed to deal with the curse of dimensionality: reducing the dimensionality [20, 21], and using parallel resources. These techniques make it feasible to do similarity search in some high-dimensional applications, but neither of them confronts the core issue of doing similarity search directly.

Finally, clustering techniques have been studied in statistics, machine learning, and database communities. Recent work in the database community includes CLARANS [27], BIRCH [36], DBSCAN [9], CLIQUE [1], and CURE [14]. These techniques have a high degree of success in identifying clusters in a very large dataset, but they do not deal with the efficiency of data search and retrieval.

3 Clindex Algorithm

Since the traditional approaches suffer from a large number of random IOs, our design objectives are 1) to reduce the number of IOs and 2) to make the IOs sequential as much as possible. To accomplish these objectives, we propose a clustering/indexing scheme.

The focus of our scheme are to

- Cluster similar data on disk to minimize disk latency for retrieving similar objects, and
- Build an index on the clusters to minimize the cost of looking up a cluster.

We call our scheme Clindex (CLustering for INDEXing).

In this paper we use a very simple clustering scheme that divides space into grids, as done in [1]. Thus, the scheme we use can be viewed as a simplified version of the algorithm in [1]. As stated earlier, our goal is not to define a new clustering algorithm, but rather to show that a very simple scheme works well in support of indexing.

Clindex consists of the following steps:

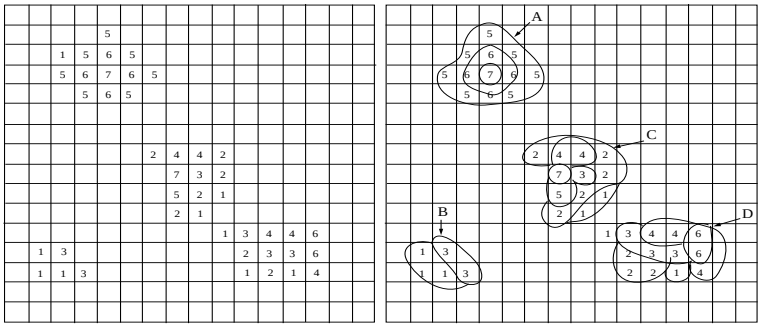
1. It divides the i^{th} dimension into 2^κ stripes. In other words, at each dimension, it chooses $2^\kappa - 1$ points to divide the dimension. (We describe how to adaptively choose these dividing points in Section 3.4.) This way, using a small number of bits (κ bits in each dimension) we can encode to which cell a feature vector belongs.

2. It groups the cells into clusters. A cell is the smallest building block for constructing clusters of different shapes. This is similar to the idea in calculus of using small rectangles to approximate polynominal functions of any degree. The finer the stripes, the smaller the cells and the finer the building blocks that approximate the shapes of the clusters (the details are described in Section 3.1). Each cluster is stored as a sequential file on disk.

3. It builds an index structure to refer to the clusters. A cell is the smallest addressing unit. A simple encoding scheme can map an object to a cell ID and retrieve the cluster it belongs to in one IO (the details are described in Section 3.2).

3.1 The CF Algorithm: Clustering Cells

To perform efficient clustering in high-dimensional spaces, we use the algorithm *cluster-forming* (CF). To illustrate the procedure, Figure 2(a) shows some points distributed on a 2D evenly divided grid. The CF algorithm works in the following way:



(a) Before Clustering (b) After Clustering

Figure 2: The Grid Before and After CF.

1. CF first tallies the *height* (the number of objects) of each cell.
2. CF starts with the cells with the highest point concentration. These cells are the peaks of the initial clusters. (In the example in Figure 2(a), we start with the cells marked 7.)
3. CF descends one unit of the height after all cells at the current height are processed. At each height, a cell can be in one of three conditions: it is not adjacent to any cluster, it is adjacent to only one cluster, or it is adjacent to more than one cluster. The corresponding actions that CF takes are
   (a) If the cell is not adjacent to any cluster, the cell is the seed of a new cluster.
   (b) If the cell is adjacent to only one cluster, we join the cell to the cluster.
   (c) If the cell is adjacent to more than one cluster, the CF algorithm invokes the *cliff-cutting* algorithm (CC) to determine to which cluster the cell belongs, or if the clusters should be combined.
4. CF terminates when the height drops to a threshold, which we call the *horizon*. The cells that do not belong to any cluster (i.e., that are below the horizon) are grouped into an *outlier* cluster and stored in one sequential file.

Figure 2(b) shows the result of applying the CF algorithm to the data presented in Figure 2(a). In contrast to how the traditional indexing schemes split the data (shown in Figure 1(a)), the clusters in Figure 2(b) follow what we call the "natural" clusters of the dataset.

The formal description of the CF algorithm is presented in Figure 3. The input to CF includes the dimension (D), the number of bits needed to encode each dimension (κ), the threshold to terminate the clustering algorithm (θ), and the dataset (P). The output consists of a set of clusters (Φ) and a heap structure (H) sorted by cell ID for indexing the clusters. For each cell that is not empty, we allocate a structure C that records the cell ID ($C.id$), the number of points in the cell ($C.\#p$), and the cluster the cell belongs to ($C.\beta$). The cells are inserted into the heap. CF is a two-pass algorithm. After the initialization step (step 0), its first pass (step 1) tallies the number of points in each cell. For each point p in the data set P, it maps the point into a cell ID by calling the function *Cell*. The function *Cell* divides each dimension into 2^κ regions. Each value in the feature vector is replaced by an integer between 0 and $2^\kappa - 1$, which depends on where the value falls in the range of the dimension. The quantized feature vector is the cell ID of the object. The CF algorithm then checks whether the cell exists in the heap (by calling the procedure $HeapFind$, which can be found in a standard algorithm book). If the cell exists, the algorithm increments the point count for the cell. Otherwise, it allocates a new cell structure, sets the point count to one, and inserts the new cell into the heap (by calling the procedure $HeapInsert$, also in standard textbooks).

In the second pass, the CF algorithm clusters the cells. In step 2 in the figure, CF copies the cells from the heap to a temporary array S, then, in steps 3 and 4 it sorts the cells in the array in descending order on the point count ($C.\#p$). In the fifth step, the algorithm checks if a cell is adjacent to some existing clusters starting from the cell with the greatest height down to the termination threshold θ. If a cell is not adjacent to any existing cluster, a new cluster β is formed in step 5.2(a). The CF algorithm records the centroid cell for the new cluster in $\beta.C$ and inserts the cluster into the cluster set Φ. If the cell is adjacent to more than one cluster, the algorithm calls the procedure CC (cliff cutting) in step 5.2(b) to determine which cluster the cell should join. (Procedure CC is described shortly.) In step 5.3, the cell joins the identified cluster (new or existing). Finally, the cells that are below the threshold are grouped into one cluster in steps 6 to 8 as outliers.

In the Cliff-Cutting (CC) procedure we need to decide to which cluster the cell belongs, or if necessary, we can merge some of the adjacent clusters. Many heuristics can be followed in this procedure. For example, we can choose a neighboring cluster with the minimal number of objects so that the cluster sizes are balanced. Or we can set an upper limit for cluster size so that there are not too many objects in a cluster. We can also try to make the "shape" of each cluster "roundish" to avoid a situation in which a cluster has a snake-like shape in a high-dimensional space. The snake-like shape is bad for a similarity search, since the distance between two objects within the cluster can be too large in that configuration. In order to keep all clusters in a desirable shape, we can compute the centroid of each, and pick the cluster whose centroid is closest to the cell. Another possible heuristic is to replicate some popular boundary cells in more than one cluster. Which heuristics are better may depend on the dataset and the recall requirement of the application.

The Cluster Forming Algorithm

- **Input**:
  - D, κ, θ, P;
- **Output**:
  - Φ; /* cluster set */
  - H; /* heap */
- **Variables**:
  - τ, ψ, S;
- **Execution Steps**:

0: Init:
$\psi \leftarrow 0$; $\Phi \leftarrow \emptyset$; $S \leftarrow \emptyset$;

1: for each $p \in P$

1.1: $\tau \leftarrow Cell(p)$; /* Map point p to a cell id */

1.2: $C \leftarrow HeapFind(H, \tau)$

1.3(a): if $(C \neq nil)$
$C.\#p \leftarrow C.\#p + 1$;

1.3(b): else
new C;
$C.id \leftarrow \tau$; $C.\#p \leftarrow 1$;
$HeapInsert(H, C)$;

2: $S \leftarrow \{C | C \in H\}$; /* S is a temp array holding a copy of all cells */

3: $Sort(S)$; /* sort cells in descending order on $C.\#p$ */

4: $i \leftarrow 0$; $C \leftarrow S[i]$;

5: while $((C \neq nil)$ and $(C.\#p \geq \theta))$

5.1: $\Psi \leftarrow FindNeighborClusters(S, C.id)$ /* Ψ contains cell C's neighboring clusters */

5.2(a): if $(\Psi = \emptyset)$ /* not attach to any cluster */
new β; /* β holds a new cluster structure */
$\beta.C \leftarrow C.id$;
$\Phi \leftarrow \Phi \cup \{\beta\}$; /* Insert the new cluster into the cluster set */

5.2(b): else If $(|\Psi| > 1)$ /* If the cell is adjacent to more than one cluster */
$\beta \leftarrow CC(C, \Psi, \Phi, H, S)$; /* CC returns which cluster cell C should join */

5.3: $C.\beta \leftarrow \beta$; /* Update to which cluster the cell belongs */

5.4: $i \leftarrow i + 1$;

6: new β; $\beta.C \leftarrow 0$; /* Group remaining cells into an outlier cluster */

7: $\Phi \leftarrow \Phi \cup \{\beta\}$;

8: for $(C = S[i]; C \neq nil; i++)$ $C.\beta \leftarrow \beta$;

Figure 3: The Cluster Forming (CF) Algorithm.

Time Complexity:

We now analyze the time complexity of the CF algorithm. Let N denote the number of objects in the dataset and M be the number of nonempty cells. Assume that it takes $O(D)$ to compute the cell ID of an object and that it takes $O(D)$ time to check whether two cells are adjacent. During the first pass of the CF algorithm, we can use a heap to keep track of the cell IDs and their heights. Given a cell ID, it takes $O(D \times log\ M)$ time to locate the cell in the heap. Therefore, the time complexity of the first phase is $O(N \times D \times logM)$.

During the second pass, the time to find all the neighboring cells is $O(min\{3^D, M\})$. The reason is that there are at most three neighboring stripes of a given cell in each dimension. We can either search all the neighboring cells, which are at most $3^D - 1$, or search all nonempty cells, which are at most M. In a high-dimensional space[2], we believe that $M << 3^D$. Therefore, the time complexity of the second phase is $O(D \times M^2)$.

The total time complexity is $O(N \times D \times log\ M) + O(D \times M^2)$. Since the clustering phase is done offline, the pre-processing time may be acceptable in light of the speed that is gained in query performance.

Remarks:

The clustering algorithm we have described often finds clusters faster than most clustering and ε-approximate nearest-neighbor algorithms. If D is large and the dataset is not uniformly distributed in the feature space, both $M << N$ and $M << 3^D$ hold. In this case, many clustering algorithms require a pre-processing time that grows exponentially with D, which is much worse than our time. If $M \approx 3^D$ or the data is uniformly distributed in the space, Clindex suffers from the curse of dimensionality as other schemes do.

The reason the other clustering algorithms are more expensive is that they are much more sophisticated and are better at finding clusters. For example, the CLIQUE algorithm ([1]) on which our is based, finds clusters in all possible reduced-dimension spaces, while our simple algorithm only searches in the full dimensional space. However, as we will see, the harder-to-find clusters often have "shapes" that are harder to exploit for storage and indexing. Thus, the "coarse" clusters found by the simple scheme are often most useful, and the extra effort of finding more refined clusters does not pay off.

3.2 The Indexing Structure

In the second step of the CF algorithm, an indexing structure is built to support fast access to the clusters generated by the CF algorithm. As shown in Figure 4, the indexing structure includes two parts: a *cluster directory* and a *mapping table*. The cluster directory keeps track of the information about all the clusters, and the mapping table is maintained to map a cell to the cluster where the cell resides.

All the objects in a cluster are stored in sequential blocks on disk, so that these objects can be retrieved by efficient sequential IOs. The cluster directory records the information about

[2]Take an image database as an example. Many image databases use feature vectors with more than 100 dimensions. Clearly, an image database stores only a tiny fraction of 3^{100} images.

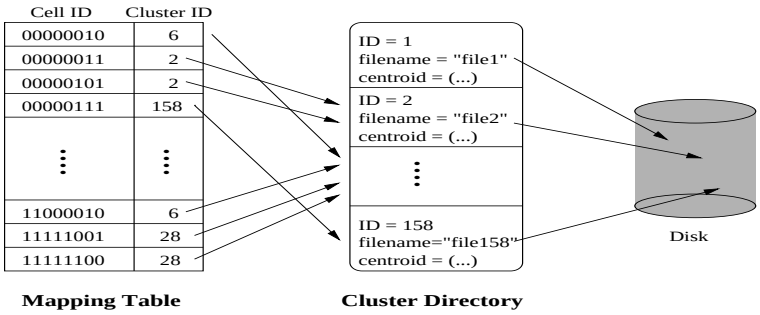


Figure 4: The Index Structure.

all the clusters, such as the cluster ID, the name of the file that stores the cluster and a flag indicating whether the cluster is an outlier cluster. The cluster's centroid, which is the center of all the cells in the cluster, is also stored. With the information in the cluster directory, the objects in a cluster can be retrieved from disk once we know the ID of the cluster.

The mapping table is used to support fast lookup from a cell to the cluster where the cell resides. Each entry has two values: a cell ID and a cluster ID. The number of entries in the mapping table is the number of nonempty cells (M), and the empty cells do not take up space in the mapping table. In the worst case, we have one cell for each object, so there are at most N cell structures. The ID of each cell is a binary code with the size $D \times \kappa$ bits, where D is the dimension and κ is the number of bits we choose for each dimension. Suppose that each cluster ID is represented as a two-byte integer. The total storage requirement for the mapping table is $M \times (D \times \kappa/8 + 2)$ bytes. In the worst case, $M = N$, and the total storage requirement for the mapping table is on the order of $O(N \times D)$. The disk storage requirement of the mapping table is comparable to that of the interior nodes in a tree index structure.

Note that the cell IDs can be sorted and stored sequentially on disk. Given a cell ID, we can easily search its corresponding entry by doing a binary search. Therefore, the number of IOs to look up a cluster is $O(log\ M)$, which is comparable the cost of doing a where-am-I search in a tree index structure.

3.3 Control Parameters

The granularity of the clusters is controlled by four parameters:

- D: the number of data dimensions or object features.
- κ: the number of bits used to encode each dimension.
- N: the number of objects.
- θ: the horizon parameter.

The number of cells is determined by parameters D and κ and can be written as $2^{D \times \kappa}$. The average number of objects of each cell is $\frac{N}{2^{D \times \kappa}}$. This means that we are dealing with two conflicting objectives. On the one hand, we do not want to have low point density, because low point density results in a large number of cells but a relatively small number of points in each

cell and hence tends to create many small clusters. On the other hand, we do not want to have densely populated cells either since having high point density results in a small number of very large clusters, which cannot help us to tell objects apart.

The value of θ affects the number and the size of the clusters. Figure 5(a) shows an example in a one-dimensional space. The horizontal axis is the cell IDs and the vertical axis the number of points in the cells. The θ value set at the t level threshold forms four clusters. The cells whose heights are below $\theta = t$ are clustered into the outlier cluster. If the threshold is reduced, both the cluster number and the cluster size increase, and the size of the outlier cluster decreases. If the outlier cluster has a relatively small number of objects, then it can fit into a few sequential disk blocks to improve its IO efficiency. On the other hand, it might be good for the outlier cluster to be relatively large, because then it can keep the other clusters well separated. In order to decide on what tradeoff is best for the two aims, we can tune the values of κ and θ properly by considering the data distribution and the desirable cluster size. Section 4.4 discusses how and when κ and θ are tuned.

3.4 Adaptive Clustering

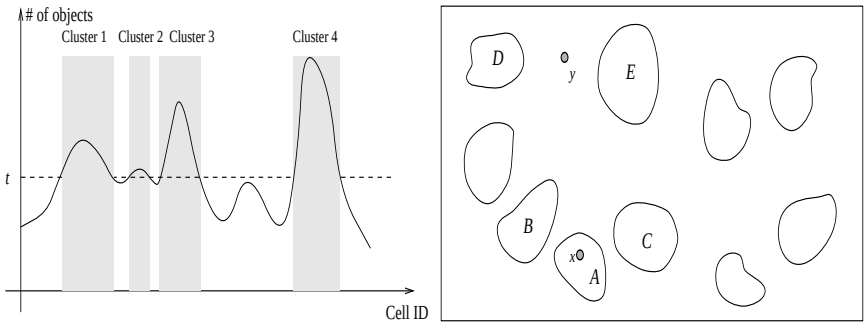
Due to the uneven distribution of the objects, it is possible that some areas are sparsely populated and others densely populated. To handle this situation, we need to be able to perform *adaptive clustering*.

Suppose we divide each dimension into 2^κ stripes. Regions that have more points, we can divide into smaller substripes. This way, we may avoid very large clusters, if this is desirable. In a way, this approach is similar to the extensible hashing scheme: for buckets that have too many points, the extensible hashing scheme splits the buckets. In our case, we can build clusters adaptively with different resolutions by choosing the dividing points carefully based on the data distribution. For example, for image feature vectors, since the luminescence is neither very high nor very low in most pictures, it makes sense to divide the luminescence spectrum coarsely at the two extremes and finely in the middle. This adaptive clustering step can be done in Step 1.1 in Figure 3. When the *Cell* procedure quantizes the value in each dimension, it can take the statistical distribution of the dataset in that dimension into consideration. This step, however, requires an additional data analysis pass before we execute the CF algorithm so that the *Cell* procedure has the information to determine how to quantize each dimension properly.

To summarize, CF is a bottom-up clustering algorithm that can approximate the cluster boundaries to any fine detail and is adaptive to data distributions. Since each cluster is stored contiguously on disk, a cluster can be accessed with much more efficient IOs than in traditional index-structure methods.

3.5 Similarity Search

Given a query object, similarity search is performed in two steps. First, Clindex maps the query object's feature vector into a binary code as the ID of the cell where the object resides. It then looks up in the mapping table the entry of the cell. The search takes the form of different actions depending on whether or not the cell is found:



(a) Clustering Example (b) Search Example

Figure 5: Illustrating Examples.

- If the cell is found, we obtain the cluster ID to which the cell belongs. We find the file name where the cluster is stored in the cluster directory. We then read the cluster from disk into memory.
- If the cell is not found, the cell must be empty. We then find the cluster closest to the feature vector by computing and comparing the distances from the centroids of the clusters to the query object. We read the nearest cluster into main memory.

If high recall is desirable, we read in more nearby clusters. After we read candidate objects into main memory, we sort and return the objects to the user according to their distances to the query object.

Remarks:

Our search can return more than one cluster whose centroid is close to the query object by checking the cluster directory. Since the number of clusters is much smaller than the number of objects in the dataset, the search for the nearest clusters can very likely be done via an in-memory lookup. If the number of clusters is very large, one may consider treating cluster centroids as points and apply clustering algorithms on the centroids. This way, one can build a hierarchy of clusters and narrow down the search space to those clusters (in a group of clusters) that are nearest to the query object. Another alternative is to precompute the k-nearest clusters and store their IDs in each cluster.

As we show in Section 4, returning the top three to four clusters can achieve very high recall (more then 90%) with very little time.

Example:

Figure 5(b) shows a 2D example of how a similarity search is carried out. In the figure, we have five clusters: A, B, C, D and E. The areas not covered by these clusters are grouped into an outlier cluster.

Suppose a user submits x as the query object. Since the cell of object x belongs to cluster A, we return the objects in cluster A and sort them according to their distances to x. If high recall is required, we return more clusters. In this case, since clusters B and C are nearby, we

also retrieve the objects in these two clusters. All the objects in clusters A, B and C are ranked based on their distances to object x, and the nearest objects are returned to the user.

If the query object is y, which falls into the outlier cluster, we first retrieve the outlier cluster. By definition, the outlier cluster stores all outliers, and hence the number of points in the outlier that can be close to y is small. We also find the two closest clusters D and E and return the nearest objects in these two clusters.

4 Evaluation

In our experiments we focused on queries of the form "find the top k most similar objects" or k-Nearest Neighbor (abbreviated as k-NN). For each k-NN query, we return the top k nearest neighbors of the query object. To establish the benchmark to measure query performance, we scanned the entire dataset for each query object to find its top 20 nearest neighbors, the "golden" results. There are at least three metrics of interest to measure the query result:

(a) Recall after X IOs: After X IOs are performed, what fraction of the k top golden results have been retrieved?

(b) Precision after X IOs: After X IOs, what fraction of the objects retrieved are among the top k golden results?

(c) Precision after $R\%$ of the top k golden objects have been found. We call this *R-precision*, and it quantifies how much "useful" work was done in obtaining some fraction of the golden results.

Due to space limitation, we focus in this paper on recall results, and comment only briefly on R-precision. In our environment, we believe that precision is not the most useful metric since the main overhead is IOs, not the number of non-golden objects that are seen.

We performed experiments with a dataset of $30,000$ images. The dataset, which was collected from two commercially available image CDs, consists of images of different content, such as landscapes, portraits, and buildings. We converted each image to a 48-dimensional feature vector by applying a wavelet transformation [6]. By setting θ to zero and κ to two (the process is discussed in Section 4.4) we obtained 261 clusters with 115 objects in each cluster on average.

In addition to using Clindex, we indexed these feature vectors using three other schemes: Equal Partition (EP), Vector Quantization (VQ) [13], and R*-tree:

- EP: To understand the role that clustering plays in Clindex, we devised a simple scheme, EP, that partitions the dataset into sequential files without performing any clustering. That is, we partitioned the dataset into cells with an equal number of images, where each cell occupies a contiguous region in the space and is stored in a sequential file. Since EP is very similar to Clindex except for the clustering, any performance differences between the schemes must be due to the clustering. If the differences are significant, it will mean that the dataset is not uniformly distributed and that Clindex can exploit the clusters.
- VQ: To evaluate the effectiveness of Clindex's CF clustering algorithm, we replaced it with a more sophisticated algorithm, and then stored the clusters sequentially as usual. The re-

placement algorithm used is VQ, a k-mean algorithm [26] implemented for clustering data in high-dimensional spaces. It has been widely used in compression and lately in indexing high-dimensional data for content-based image retrieval [31, 32].

- R*-tree: Tree structures are often used for similarity searches in multidimensional space. To compare, we used the R*-tree structure implemented by Katayama and Satoh [22]. We studied the IO cost and recall of using a tree-like structure to perform similarity search approximately. Although R*-tree may not be the best implementation of a tree structure, we believe it is a good representative. Note that the Clindex-R*-tree comparison will not be "fair" since R*-tree performs no offline analysis (i.e., no clustering is done in advance). Thus, the results will only be useful to quantify the potential gains that the offline analysis gives us, compared to traditional tree-based similarity searching.[3]

As discussed in Section 3.5, Clindex always retrieves the next cluster whose centroid is closest to the query object. We also added this intelligence to both VQ and EP to improve their recall. For R*-tree, however, we did not add this optimization.

To measure recall, we used the cross-validation technique [26] commonly employed to evaluate clustering algorithms. We ran each test ten times; each time we set aside 100 images as the test images and used the remaining images to build indexes. We then used these set-aside images to query the database built under four different index structures. We produced the average query recall for each run by averaging the recall of 100 image queries. We then took an average over 10 runs to obtain the final recall.

We were interested in finding out the effects of the following two factors on recall: (1) clustering algorithms and (2) block (cluster) size. We first collected the recall for 20-NN queries. In Section 4.3, we present the recall for k-NN, where $k = 1$ to 20.

4.1 Recall versus Clustering Algorithms

Figure 6 compares the recall of Clindex, VQ, EP and R*-tree. In this experiment, all schemes divided the dataset into about 256 clusters. Given one IO to perform, Clindex returns the objects in the nearest cluster, which gives us an average of 62% recall (i.e., a return of 62% of the top 20 golden results). After we read three more clusters, the accumulated recall of Clindex increases to 90%. If we read more clusters, the recall still increases but at a much slower pace. After we read 15 clusters, the recall approaches 100%. That is, we can selectively read in 6% ($\frac{15}{261} = 6\%$) of the data in the dataset to obtain almost all top 20 golden results.

The EP scheme achieves much lower recall than Clindex. It starts with 30% recall after the first IO is completed and slowly progresses to 83% after 30 IOs. The VQ structure, although it does better than the EP scheme, still lags behind Clindex. It achieves 35% recall after one IO, and the recall does not reach 90% until after ten IOs. The recall of Clindex and VQ converge after 20 IOs. Finally, R*-tree suffers from the lowest recall.

[3] A scheme like R*-tree could be modified to pre-analyze the data and build a better structure. The results would be presumably similar to the Clindex results, although with a much more complex algorithm. We did not develop the modified R*-tree scheme and hence did not verify our hypothesis that performance would be equivalent.

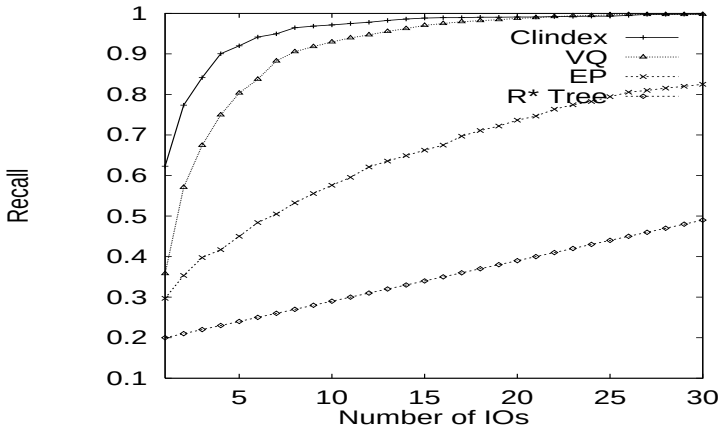


Figure 6: Recall of Four Schemes (20-NN).

From Figure 6, we can draw the following conclusions:

1. Clustering indeed helps improve recall when a dataset is not uniformly distributed in the space. This is shown by the recall gap between Clindex and EP.

2. Clindex achieves higher recall than VQ, EP and R*-tree because Clindex adjusts to wide variances of cluster shapes and sizes. VQ and the other algorithms are forced to bound a cluster by linear functions, e.g., a cube in 3-D space. Clindex is not constrained in this way. Thus, if we have an odd cluster shape (e.g., with tentacles), Clindex will conform to the shape, while VQ will either have to pick a big space that encompasses all the "tentacles," or will have to select several clusters, each for a portion of the shape, and will thereby inadvertently break some "natural" clusters as we illustrated in Figure 1(a). It is not surprising that the recall of VQ converges to that of Clindex after a number of IOs because VQ eventually pieces together its "broken" clusters.

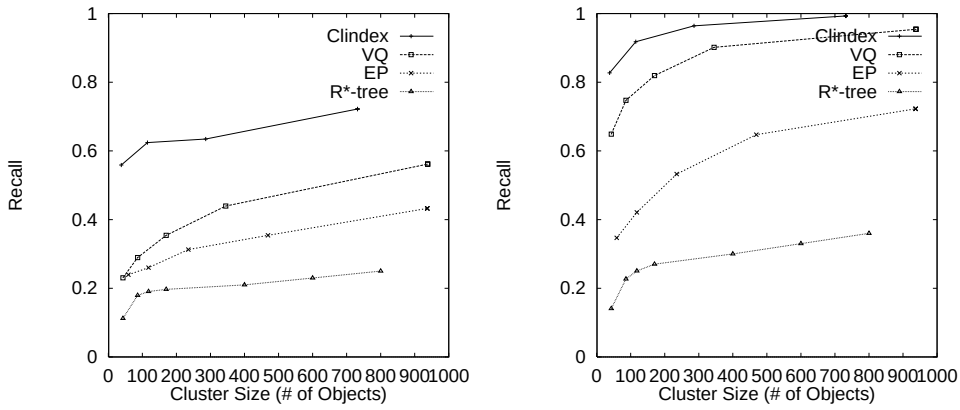
3. Using Clindex to approximate an exact similarity search requires reading just a fraction of the entire dataset. The performance is far better than that achieved by performing a sequential scan on the entire dataset.

4. Clindex, VQ and EP enjoy a boost in recall in their first few additional IOs since they always retrieve the next cluster whose centroid is closest to the query object. R*-tree, conversely, does not store centroid information and cannot selectively retrieve blocks to boost its recall in its first few IOs. (We have discussed the shortcomings of tree structures in Section 2 in detail.)

	Recall	60%	70%	80%	90%	$\approx$ 100%
	# of Golden Objects Retrieved	12	14	16	18	20
Clindex	*# of Objects Retrieved*	115	230	345	460	1,725
	R-Precision	10.44%	6.09%	4.64%	3.91%	1.16%
R-tree*	*# of Objects Retrieved*	2,670	3,430	4,090	4,750	5,310
	R-Precision	0.45%	0.41%	0.39%	0.379%	0.377%

Table 1: *R*-Precision of 20-NN

Figure 6 shows that in retrieving the same amount of data, Clindex has higher recall than R*-tree and other schemes. Put another way, Clindex also enjoys higher *R*-precision because retrieving the same number of objects from disk gives Clindex more golden results.



(a) After One IO

(b) After Five IOs

Figure 7: Recall versus Cluster Size.

Since a tree structure is typically designed to use a small block size to achieve high precision, we tested R*-tree and compared its R-precision with that of Clindex. It took R*-tree 267 IOs on average to complete an exact similarity search. Table 1 shows that the R-precision of Clindex is more than ten times higher than that of R*-tree under different recall values.

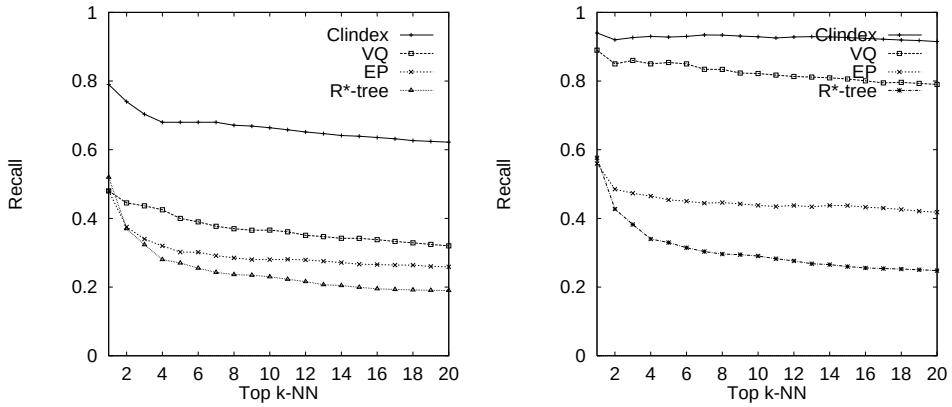
This result, though surprising, is consistent with that of many studies [22, 33]. The common finding is that when the data dimension is high, tree structures fail to divide points into neighborhoods and are forced to access almost all leaves. Therefore, the argument for using a small block size to improve precision becomes weaker as the data dimension increases.

4.2 Recall versus Cluster Size

We define cluster size as the average number of objects in a cluster. To see the effects of the cluster size on recall, we collected recall values at different cluster sizes for Clindex, VQ, CF and R*-tree. Note that the cluster size of Clindex is determined by the selected values of κ and θ. We thus set four different κ and θ combinations to obtain recall values for four different cluster sizes. For EP, VQ and R*-tree, we selected cluster (block) sizes that contain from 50 up to 900 objects. Figure 7 depicts the recall (y-axis) of the four schemes for different cluster sizes (x-axis) after one IO and after five IOs are performed.

Figure 7(a) shows that given the same cluster size, Clindex has a higher recall than VQ, EP and R*-tree. The gaps between the schemes widen as the cluster size increases. We believe that the larger the cluster size, the more important a role the quality of the clustering algorithms plays. Clindex enjoys significantly higher recall than VQ, EP and R*-tree because it captures the clusters better than the other three schemes.

Figure 7(b) shows that Clindex still outperforms VQ, EP and R*-tree after five IOs. Clindex, VQ and EP enjoy better improvement in recall than R*-tree because, again, Clindex, VQ and EP always retrieve the next cluster whose centroid is nearest to the query object. On the other hand, a tree structure does not keep the centroid information, and the search in the neighborhood can be random and thus suboptimal. We believe that adding the centroid information to the



(a) After One IO (b) After Five IOs

Figure 8: Recall of k-NN.

leaves of the tree structures can improve its recall.

4.3 Recall versus k-NN

So far we measured only the recall for returning the top 20 nearest neighbors. In this section we present the recall when returning the top $n = 1$ to 19 nearest neighbors, after one and five IOs are performed. In these experiments, we partitioned the dataset into about 256 clusters (blocks) for all schemes (i.e., all schemes have about the same cluster size).

Figures 8(a) and 8(b) present the recall of the four schemes after one IO and five IOs are performed, respectively. The x-axis in the figures represents the number of nearest neighbors requested and the y-axis the recall. For instance, when one and only one nearest object is requested, Clindex returns the nearest object 79% of the time after one IO and 95% of the time after five IOs.

Both figures show that the recall gaps between schemes are insensitive to how many nearest neighbors are requested. We believe that once the nearest object can be found in the returned cluster(s), the conditional probability that additional nearest neighbors can also be found in the retrieved cluster(s) is high. Of course, for this conclusion to hold, the number of objects contained in a cluster must be larger than the number of nearest neighbors requested. (In our experiment, an average cluster contains 115 objects and we tested up to 20 golden results.) This leads us to the following discussion on how to tune Clindex's control parameters to form clusters of a proper size.

4.4 The Effects of the Control Parameters

We experimented with different values of θ and κ to form clusters. Since the size of our dataset is much smaller than the number of cells ($2^{D \times \kappa}$), many cells are occupied by only one object. When we set $\theta \geq 1$, most points fell into the outlier cluster. We thus set θ to zero.

To test the κ values, we started with $\kappa = 2$. We increased κ by increments of one and checked the effect on the recall. Figure 9 shows that the recall with respect to the number of IOs decreases when κ is set beyond two. This is because in our dataset $D >> log\ N$, and using a κ that is too large spreads the objects apart and thereby leads to the formation of too many small clusters. By dividing the dataset too finely one loses the benefit of large block size for sequential IOs and hence doing so is not beneficial.

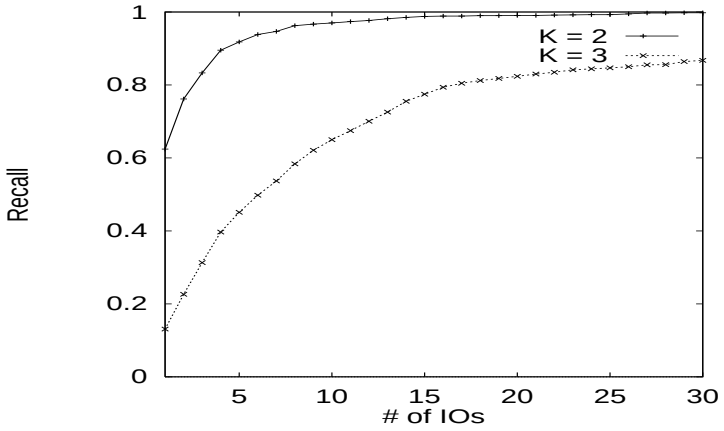


Figure 9: Recall versus κ.

Although the proper values of κ and θ are dataset dependent, we empirically found the following rules of thumb to be useful for finding good starting values:

- Choose a reasonable cluster size: The cluster must be large enough to take advantage of sequential IOs. It must also contain enough objects so that if a query object is near a cluster, then the probability is high that a significant number of nearest neighbors of the query object are in the cluster. On the other hand, a cluster should not be so large as to prolong query time unnecessarily. According to our experiments, increasing the cluster size to where the cluster contains more than 300 objects (see Figure 7) does not further improve recall significantly. Therefore, a cluster of about 300 objects is a reasonable size.
- Determine the desired number of clusters: Once the cluster size is chosen, one can compute how many clusters the dataset will be divided into. If the number of clusters is too large to make the cluster table fit in main memory, we may either increase the cluster size to decrease the number of clusters or consider building an additional layer of clusters.
- Adjust κ and θ: Once the desired cluster size is determined, we pick the starting values for κ and θ. The κ value must be at least two so that the points are separated, and the suggested θ is one so that the clusters are separated. After running the clustering algorithm with the initial setting, if the average cluster is smaller than the desired size, we set θ to zero to combine small clusters. If the average cluster is larger than the desired size, we can increase either κ or θ until the we obtain roughly the desired cluster size.

4.5 Summary

In summary, our experimental results show that:

1: Using a large block size is good for both IO efficiency and recall.

2: Employing a better clustering algorithm improves recall.

3: Providing additional information such as the centroids of the clusters helps prioritize the retrieval order of the clusters and hence improves the search efficiency.

5 Conclusions

In this paper, we presented Clindex: a new paradigm for performing similarity search in high-dimensional spaces to avoid the dimensionality curse. We cluster similar objects on disk and perform a similarity search by finding the clusters near the query object. This approach improves the IO efficiency by clustering and retrieving relevant information sequentially on and from the disk. Its pre-processing cost is linear in D and polynomial in N and its query cost is independent of D.

Experiments showed that Clindex typically can achieve 90% recall after performing just a few sequential IOs. Clindex's recall is much higher than that of some traditional index structures. Through experiments we also learned that Clindex works well because it uses large blocks, finds clusters more effectively, and searches the neighborhood more intelligently by using the centroid information. These design principles can also be employed by other schemes to improve their search performance. (For example, one may replace the clustering example in this paper with one that is more suitable for a particular dataset.) We believe that for many high-dimensional datasets that exhibit clustering patterns, e.g., documents and images, Clindex is an attractive approach to support approximate similarity search both efficiently and effectively.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proceedings of ACM SIGMOD*, June 1998.

[2] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Proceedings of the 5th SODA*, pages 573–82, 1994.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proceedings of ACM SIGMOD*, May 1990.

[4] S. Berchtold. The X-Tree: An index structure for high-dimensional data. *Proceedings of the 22nd VLDB*, August 1996.

[5] S. Brin and H. Garcia-Molina. Copy detection mechanisms for digital documents. *Proceedings of ACM SIGMOD*, May 1995.

[6] E. Chang, J. Wang, C. Li, and G. Wiederhold. RIME - a replicated image detector for the world-wide web. *Proc. of SPIE Symposium of Voice, Video, and Data Communications*, November 1998.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: An efficient access method for similarity search in metric spaces. *Proceedings of the 23rd VLDB*, August 1997.

[8] K. Clarkson. An algorithm for approximate closest-point queries. *Proceedings of the 10th SCG*, pages 160–64, 1994.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[10] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, and et al. Query by image and video content: the QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[11] H. Garcia-Molina, S. Ketchpel, and N. Shivakumar. Safeguarding and charging for information on the internet. *Proceedings of ICDE*, 1998.

[12] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 1999.

[13] A. Gersho and R. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, 1991.

[14] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Proceedings of ACM SIGMOD*, June 1998.

[15] A. Gupta and R. Jain. Visual information retrieval. *Comm. of the ACM*, 40(5):69–79, 1997.

[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. of ACM SIGMOD*, June 1984.

[17] P. Indyk, A. Gionis, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th VLDB*, September 1999 (to appear).

[18] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Proceedings of the 30th STOC*, pages 604–13, 1998.

[19] P. Indyk, R. Motwani, and P. Raghavan. Locality-preserving hashing in multidimensional spaces. *Proceedings of the 29th STOC*, pages 618–25, 1997.

[20] W. Johnson and J. Lindenstrauss. Extension of lipschitz mapping into hilbert space. *Contemporary Mathematics*, 24:189–206, 1984.

[21] K. V. R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. *Proceedings of ACM SIGMOD*, pages 166–76, 1998.

[22] N. Katayama and S. Satoh. The SR-Tree: An index structure for high-dimensional nearest neighbor queries. *Proceedings of ACM SIGMOD*, May 1997.

[23] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proceedings of the 29th STOC*, 1997.

[24] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proceedings of the 30th STOC*, pages 614–23, 1998.

[25] K.-L. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: an index structure for high-dimensional data. *VLDB Journal*, 3(4), 1994.

[26] T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[27] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proceedings of the 20th VLDB*, September 1994.

[28] J. T. Robinson. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. *Proceedings of ACM SIGMOD*, April 1981.

[29] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *ACM Sigmod*, pages 71–79, 1995.

[30] Y. Rubner, C. Tomasi, and L. Guibas. Adaptive color-image embedding for database navigation. *Proceedings of the the Asian Conference on Computer Vision*, January 1998.

[31] J. Z. Wang, G. Wiederhold, O. Firschein, and S. X. Wei. Wavelet-based image indexing techniques with partial sketch retrieval capability. *Proceedings of the 4th ADL*, May 1997.

[32] J. Z. Wang, G. Wiederhold, O. Firschein, and S. X. Wei. Content-based image indexing and searching using daubechies' wavelets. *International Journal of Digital Libraries*, 1(4):311–28, 1998.

[33] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *Proceedings of the 24th VLDB*, pages 194–205, 1998.

[34] D. A. White and R. Jain. Similarity indexing: Algorithms and performance. *Proc. SPIE Vol.2670, San Diego*, 1996.

[35] D. A. White and R. Jain. Similarity indexing with the SS-Tree. *Proceedings of the 12th ICDE*, Feb. 1996.

[36] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proceedings of ACM SIGMOD*, June 1996.